# Thread-Modular Model Checking with Iterative Refinement *

Wenrui Meng[1,2,3], Fei He[1,2,3], Bow-Yaw Wang[4], and Qiang Liu[1,2,3]

[1] School of Software, Tsinghua University
[2] Tsinghua National Laboratory for Information Science and Technology (TNList)
[3] Key Laboratory for Information System Security, MOE, China
[4] Institute of Information Science, Academia Sinica

**Abstract.** Thread-modular analysis is an incomplete compositional technique for verifying concurrent systems. The heuristic works rather well when there is limited interaction among system components. In this paper, we develop a refinement algorithm that makes thread-modular model checking complete. Our algorithm refines abstract reachable states by exposing local information through auxiliary variables. The experiments show that our complete thread-modular model checking can outperform other complete compositional reasoning techniques.

## 1 Introduction

Compositional reasoning is a promising technique to alleviate the state explosion problem in model checking [2, 17]. In compositional reasoning, one decomposes a verification problem into simpler subproblems and solves each subproblem one at a time. By the soundness of decomposition, the verification problem is solved if all subproblems are solved. Soundness of decomposition apparently depends on the underlying computation model. In this paper, we are interested in verifying invariant properties on shared-memory interleaving systems.

A shared-memory interleaving system consists of several components. Each component has two types of variables. Global variables are accessible to every component in the system. Local variables, on the other hand, are only accessible to the defining component. At any moment, exactly one component is active. Inactive components do not perform any computation and hence keep their local variables unchanged. Global variables may be updated by the active component nonetheless. In such systems, global variables are used for communication among components. Given a predicate on system states, the invariant checking problem is to verify whether the given predicate holds on every reachable states.

Two compositional techniques for the invariant checking problem on shared-memory interleaving systems are known. In thread-modular reasoning [9, 5],

* This work was supported in part by the Chinese National 973 Plan under grant No. 2010CB328003, the NSF of China under grants No. 60903030, the Tsinghua University Initiative Scientific Research Program.

one computes an over-approximation of reachable system states by intersecting reachable component states of all components. In order to compute reachable component states of a designated component, one disregards local variables of other components and computes an abstract model of global variables. The designated component is then composed with the abstract model to compute its reachable component states. The effectiveness of thread-modular reasoning depends on the abstraction. If the abstraction of global variables is able to establish the property, one concludes the verification. Otherwise, one reports that the verification is inconclusive.

The (in)effectiveness problem in thread-modular reasoning is solved in the second compositional technique called local proof [4]. In local proof, one still requires reachable states of each component. Reachable component states however are computed by early quantification of reachable system states. Abstract models for global variables are hence not needed. Moreover, techniques have been developed to refine reachable component states. Local proof is hence a complete compositional technique for shared-memory interleaving systems.

Although both techniques compute reachable component states and use the intersection as an over-approximation of reachable system states, we would like to point out a subtle difference between them. In thread-modular reasoning, one constructs an abstract model for global variables. Reachable component states are then computed via the abstract model. In local proof, on the other hand, reachable component states are computed by quantifying out inaccessible local variables during the exploration of reachable system states. Since no abstraction is deployed during the exploration of component states, reachable component states in local proof are more precise than those of thread-modular reasoning. On the other hand, the computation of reachable component states in local proof can be more expensive then thread-modular reasoning due to no abstraction. One wonders whether an efficient yet complete compositional technique exists for such systems.

Inspired by the refinement in local proof, we propose a complete thread-modular model checking algorithm for the invariant checking problem on shared-memory interleaving systems. Our technique contains two phases. At the verification phase, we apply thread-modular reasoning to the verification problem. If the compositional technique suffices to conclude the verification, we are done. Otherwise, our technique moves to the refinement phase. In the other phase, we adopt ideas from local proof and expose information about local variables during refinement. More precisely, we identify local variables that can refine the approximation to reachable system states. Such information is then exposed to other components by adding global variables. When our technique returns to the verification phase, added variables will induce a refined abstract model for global variables. Efficiency of thread-modular reasoning and effectiveness of local proof are thus attained by our proposed technique.

We implement our thread-modular model checking with iterative refinement algorithm on NuSMV, and compare with other algorithms in five examples. Due to its aggressive abstraction, thread-modular reasoning fails to verify all examples but the bakery algorithm. Our new technique performs better than local proof in our examples. In several examples, our compositional technique outperforms monolithic techniques in orders of magnitude. Our preliminary experimental results suggest that an efficient yet complete compositional technique is indeed possible for shared-memory interleaving systems.

### 1.1 Related Work

In 1976, Owicki and Gries proposed some non-interference proof rules for parallel programs in their work [15]. Chandy and Misra [13] and Jones [9] [10] extended those rules with interference to introduce thread-modular reasoning. To make thread-modular model checking automatic, the environment is automatically generated [5] according to the interactions of the programs. Henzinger et al. [7] [8] improved the original thread-modular model checking and made it complete for safety property verification on finite state systems. In their approaches, each thread is initialized as true and is then iteratively refined by addition of new predicates, and the guarantee of each thread is initialized as false and is successively refined by considering abstract of current thread and guarantees of other threads. Recently, Gu et al. [6] attempted to improve the generation of environment assumptions with horn logic deductive rule. Malkis et al. [12] proposed a technique, called thread-modular counterexample guided abstraction refinement, which computes reachable states with cartesian abstraction. A refinement step was involved to eliminate the infeasible states by excluding them from the cartesian product. But this approach directly computes the reachable states for all processes of concurrent system in an explicit way.

Another interesting branch for concurrent system verification is based on the inductive invariant rule. The invisible invariants method [16] [1] generated quantified invariants for parameterized protocols by analyzing reachable states of a small instance; however, it is incomplete for some protocols. Absorbing the completeness theory of [15] and [11], Namjoshi extended the inductive invariant to non-interference invariant named split invariant [14]. Based on split invariant, Cohen and Namjoshi proposed a local proof algorithm for global safety properties of concurrent systems and used refinement procedure to make the verification complete [4].

The remainder of this paper proceeds as follows. Section 2 gives basic definitions. It is followed by a brief overview of thread-modular reasoning in Section 3. Our technical contribution is presented in Section 4. Section 5 gives our experimental results. We conclude our presentation in Section 6.

## 2 Preliminary

We assume a fixed set $V$ of typed variables. A *state over* $W \subseteq V$ is a valuation for the variables in $W$. The set of states over $W \subseteq V$ is denoted by $St[W]$. For $W \subseteq V$ and $s \in St[V]$, the *projection of* $s$ *on* $W$ (written $s \downarrow_W$) is a state over $W$ that $s \downarrow_W (w) = s(w)$ for every $w \in W$. Let $W \subseteq V$, we write $St[V] \downarrow_W$

to indicate the set $St[W] = \{s\downarrow_W \| \forall s \in St[V]\}$. Given $St[W]$ and $St[X]$, their join is $St[W \cup X] = \{s | s\downarrow_W \in St[W] \text{ and } s\downarrow_X \in St[X]\}$ which is denoted by $St[W] \bowtie St[X]$. A *predicate over* $St[V]$ is a function from $St[V]$ to the Boolean domain $\mathbb{B}$. Given a state $s \in St[V]$ and a predicate $\phi$ over $St[V]$, we say $s$ *satisfies* $\phi$ (written $s \models \phi$) if $\phi(s) = \top$. For any predicate $\phi$ over $St[V]$, define $[\![\phi]\!] = \{s \in St[V] : \phi(s)\}$. That is, $[\![\phi]\!]$ consists of states that satisfy $\phi$. For $W \subseteq V$ and a predicate $\phi$ over $St[V]$, define the predicate $\phi\downarrow_W$ over $St[W]$ to be that for any $t \in St[W]$,

$$\phi\downarrow_W(t) = \top \text{ if and only if there is an } s \in St[V] \text{ with } \phi(s) = \top \text{ and } s\downarrow_W = t.$$

A *process* $P = \langle X, L, I, T\rangle$ is a quadruple where $X \subseteq V$ is the set of *global variables*, $L \subseteq V$ the set of *local variables* disjoint from $X$, $I$ the *initial predicate* over $St[X \cup L]$, and $T$ the *transition predicate* over $St[X \cup L] \times St[X \cup L]$. Let $s, s' \in St[X \cup L]$. We say $s$ is *initial* if $I(s) = \top$. If $T(s, s') = \top$, we say $s$ is a *predecessor* of $s'$ and $s'$ a *successor* of $s$. A *trace* $\tau$ is a sequence of states $s^0, s^1, \ldots, s^n$ such that $I(s^0) = \top$ and $T(s^i, s^{i+1}) = \top$ for $0 \le i < n$. The set of traces of $P$ is denoted by $Tr[P]$. A state $s$ is *reachable* in $P$ if there is a trace $\tau = s^0, s^1, \ldots, s^n \in Tr[P]$ such that $s^n = s$. The set of states reachable in $P$ is denoted by $Re[P]$. Let $\pi$ be a predicate over $St[X \cup L]$. We say $P$ satisfies $\pi$ (written $P \models \pi$) if $s \models \pi$ for every $s \in Re[P]$.

Let $P_j = \langle X, L_j, I_j, T_j\rangle$ be processes for $j = 0, 1$ where $L_0$ and $L_1$ are disjoint. Let $W_j = X \cup L_j \subseteq V$ for $j = 0, 1$. The *composition* of $P_0$ and $P_1$ (written $P_0 \| P_1$) is a process $\langle X, L, I, T\rangle$ where

- $L = L_0 \cup L_1$;
- $I(s) = \top$ if $I_0(s\downarrow_{W_0}) = \top$ and $I_1(s\downarrow_{W_1}) = \top$;
- $T(s, s') = \top$ if
  - $T_0(s\downarrow_{W_0}, s'\downarrow_{W_0}) = \top$ and $s\downarrow_{L_1} = s'\downarrow_{L_1}$; or
  - $T_1(s\downarrow_{W_1}, s'\downarrow_{W_1}) = \top$ and $s\downarrow_{L_0} = s'\downarrow_{L_0}$.

That is, exactly one process updates the global variables and its local variables; the other process stutters in a transition of the composition. It is straightforward to see that the composition is associative. $P_1 \| P_2 \| \cdots \| P_N$ is thus well-defined for $N \ge 2$.

## 3 Thread-Modular Reasoning

**Definition 1.** *Let $P = \langle X, L, I, T\rangle$ be a process. The* guarantee *of $P$ is a process $G(P) = \langle X, \emptyset, I_G, T_G\rangle$ where $I_G = I\downarrow_X$ and $T_G$ is a predicate over $St[X] \times St[X]$ such that*

$$T_G(t, t') = \top \textit{ if } \exists s, s' \in St[X \cup L] \textit{ with } T(s, s') = \top, s\downarrow_X = t, \textit{and } s'\downarrow_X = t'.$$

The main process of thread-modular model checking is shown in Algorithm 1. It first computes the reachable component states $\overline{R}_j$ for each process $P_j$, then computes the reachable system states $\overline{R}$ by joining reachable component states

**Input**: $P_j = \langle X, L_j, I_j, T_j \rangle$ : a process for $1 \le j \le N$; $\pi$ : a predicate over $St[X \cup L_1 \cup \cdots \cup L_N]$

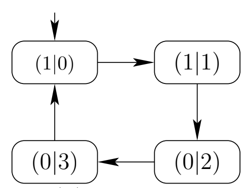
**Output**: "*PASS*" or "*UNKNOWN*"

$error \leftarrow [\![\neg\pi]\!]$;<br>
**foreach** $j = 1, \ldots, N$ **do**<br>
    $\overline{R}_j \leftarrow Re[G(P_1) \| \cdots \| G(P_{j-1}) \| P_j \| G(P_{j+1}) \| \cdots \| G(P_N)]$;<br>
**end**<br>
$\overline{R} \leftarrow \overline{R}_1 \bowtie \overline{R}_2 \bowtie \cdots \bowtie \overline{R}_N$;<br>
**if** $\overline{R} \cap error = \emptyset$ **then**<br>
    **return** *PASS*;<br>
**else**<br>
    **return** *UNKNOWN*;

**Algorithm 1**: Thread-Modular Model Checking

global $x$: boolean initially $x = 1$<br>
loop forever<br>
$\left[\begin{array}{l} l = 0 : \text{Non-Critical} \\ l = 1 : \text{request } x \\ l = 2 : \text{Critical} \\ l = 3 : \text{release } x \end{array}\right.$

(a) Textual




(b) Graphical

**Fig. 1.** MUX-SEM$_k$

of all processes. Apparently, $\overline{R}$ is an over-approximation of the reachable system states, so it can report "PASS" when there is no error state in $\overline{R}$. Otherwise, it cannot make any conclusion.

*Example 1.* Consider a simple solution to the mutual exclusion problem in Fig. 1. In the figure, $N$ processes attain mutual exclusion by the semaphore $x$. Each process requests $x$ before entering the critical section, and releases $x$ after leaving the critical section. Assume there are two processes $P_1$ and $P_2$. We use $P_j.l$ to denote the local variable $l$ in process $P_j$ where $j \in \{1, 2\}$. Each state in Fig. 1 (b) is marked with the corresponding valuation for all variables, where before the separator $|$ is the valuation for global variables, and after the separator $|$ is the valuation for local variables. Mutual exclusion is specified as $\pi : \neg((P_1.l = 2 \vee P_1.l = 3) \wedge (P_2.l = 2 \vee P_2.l = 3))$. We have the guarantee $G(P_j) = \langle \{x\}, \emptyset, I_j, T_j \rangle$ where $j \in \{1, 2\}$, $I_j(s)$ is $s(x) = 1$, and $T_j(s, s')$ is $\top$. Hence

$$\overline{R}_1 = Re[G(P_1) \| P_2] = \{s : s(x) \in \{0, 1\} \text{ and } s(P_1.l) \in \{0, 1, 2, 3\}\}$$
$$\overline{R}_2 = Re[P_1 \| G(P_2)] = \{s : s(x) \in \{0, 1\} \text{ and } s(P_2.l) \in \{0, 1, 2, 3\}\}$$

Thus,

$$\overline{R} = \overline{R}_1 \bowtie \overline{R}_2 = \{s : s(x) \in \{0, 1\}, s(P_1.l) \in \{0, 1, 2, 3\}, \text{ and } s(P_2.l) \in \{0, 1, 2, 3\}\}.$$

Since $\overline{R} \cap [\![\neg\pi]\!] \neq \emptyset$, Algorithm 1 reports "*UNKNOWN*."

## 4 Iterative Refinement

Let $P = \langle X, L, I, T\rangle$ be a process, $l \in L$, and $S$ a set of states. We say $l$ is an *essential variable* of $P$ with respect to $s \in S$ if there is a $t \in St[X \cup L]$ such that

- $t \notin S$;
- $s(l) \neq t(l)$; and
- $s(v) = t(v)$ for every $v \in (X \cup L) \setminus \{l\}$.

In other words, a local variable is essential with respect to a state set if its value signifies the membership of the given state set.

Let $l$ be an essential variable with respect to $s \in S$. Define the *essential predicate* $\chi_l^s$ for $l$ with respect to $s \in S$ by

$$\chi_l^s(t) = \top \text{ if } t(l) = s(l).$$

Two essential predicate $\chi_l^s$ and $\chi_m^t$ are *distinct* if either $l$ is different from $m$ or $s(l) \neq t(m)$.

*Example 2.* In Example 1, observe that

$$\overline{R} \cap [\![\neg\pi]\!] = \{s : s(x) \in \{0,1\}, s(P_1.l) \in \{2,3\}, \text{ and } s(P_2.l) \in \{2,3\}\}.$$

Let us consider the state $s_0 \in \overline{R} \cap [\![\neg\pi]\!]$ that $s_0(x) = 0$, $s_0(P_1.l) = s_0(P_2.l) = 2$. Define $t_0 \in St[\{x, P_1.l, P_2.l\}]$ where $t_0(x) = 0$, $t_0(P_1.l) = 1$, and $t_0(P_2.l) = 2$. Then $t_0 \notin \overline{R} \cap [\![\neg\pi]\!]$, $s_0(P_1.l) \neq t_0(P_1.l)$, and $s_0(v) = t_0(v)$ for $v \in \{x, P_2.l\}$. Hence $P_1.l$ is an essential variable of $P_1$ with respect to $s_0$. The essential predicate $\chi_{P_1.l}^{s_0}$ for $P_1.l$ is hence

$$\chi_{P_1.l}^{s_0}(t) = \top \text{ if } t(P_1.l) = 2.$$

Similarly, consider the state $s_1 \in \overline{R} \cap [\![\neg\pi]\!]$ that $s_1(x) = 0$, $s_1(P_1.l) = 3$, $s_1(P_2.l) = 2$. Define $t_1(x) = 0$, $t_1(P_1.l) = 1$, and $t_1(P_2.l) = 2$. Then $P_1.l$ is an essential variable of $P_1$ with respect to $s_1$. The essential predicate $\chi_{P_1.l}^{s_1}$ for $P_1.l$ is therefore

$$\chi_{P_1.l}^{s_1}(t) = \top \text{ if } t(P_1.l) = 3.$$

**Definition 2.** *Let $P = \langle X, L, I, T\rangle$ be a process and $\Psi$ a set of predicates. Define $W = X \cup L$. The* augmented process $A(P, X_A, X_\Psi, \Psi) = \langle X \cup X_A, L, I_A, T_A\rangle$ *of $P$ with $\Psi$ is defined by*

- *$X_\Psi = \{u_\chi \in V : \chi \in \Psi\}$ is the set of auxiliary variables with respect to $\Psi$;*
- *$X_\Psi \subset X_A$ and $X_A - X_\Psi$ is other processes' auxiliary variables;*
- *$I_A(s) = \top$ if $I(s\downarrow_W) = \top$ and $s(u_\chi) = \chi(s)$ for every $\chi \in \Psi$;*
- *$T_A(s, s') = \top$ if $T(s\downarrow_W, s'\downarrow_W) = \top$, $s(u_\chi) = \chi(s)$, $s'(u_\chi) = \chi(s')$ for every $\chi \in \Psi$ and $s'(v) = s(v)$ for every $v \in X_A - X_\Psi$.*

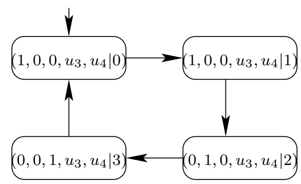
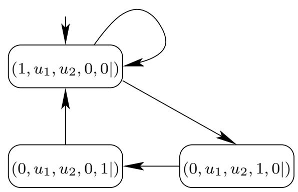
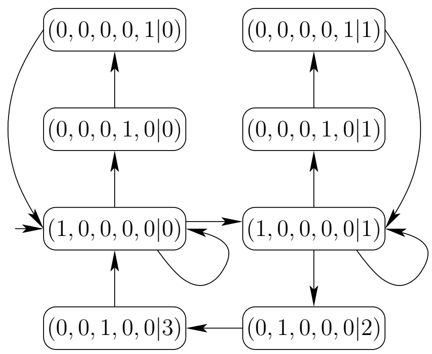



**Fig. 2.** $A(\text{MUX-SEM}_1, \{\chi^{s_0}_{P_1.l}, \chi^{s_1}_{P_1.l}\})$




(a) $G(A(\text{MUX-SEM}_2, \{\chi^{s_0}_{P_1.l}, \chi^{s_1}_{P_1.l}\}))$




(b) $A(\text{MUX-SEM}_1, \{\chi^{s_0}_{P_1.l}, \chi^{s_1}_{P_1.l}\})\|$ $G(A(\text{MUX-SEM}_2, \{\chi^{s_0}_{P_2.l}, \chi^{s_1}_{P_2.l}\}))$

**Fig. 3.** Example 3

*Example 3.* Recall the essential predicates $\chi^{s_0}_{P_1.l}$ and $\chi^{s_1}_{P_1.l}$ from Example 2. Let $\Psi = \{\chi^{s_0}_{P_1.l}, \chi^{s_1}_{P_1.l}\}$. Denote the auxiliary variables for $\chi^{s_0}_{P_1.l}$, $\chi^{s_1}_{P_1.l}$, $\chi^{s_0}_{P_2.l}$, and $\chi^{s_1}_{P_2.l}$ as $u_1, u_2, u_3, u_4$ respectively. Fig. 2 shows the augmented process $A(\text{MUX-SEM}_1, X_A, X_\Psi, \Psi)$, where $X_\Psi = \{u_1, u_2\}$ and $X_A = \{u_1, u_2, u_3, u_4\}$, and Fig. 3 shows its composition with the guarantee of augmented $\text{MUX-SEM}_2$. Thus

$$\overline{R}_1 = \left\{ \begin{array}{llll} (1,0,0,0,0|0), & (1,0,0,0,0|1), & (0,1,0,0,0|2), & (0,0,1,0,0|3), \\ (0,0,0,1,0|0), & (0,0,0,0,1|0), & (0,0,0,1,0|1), & (0,0,0,0,1|1) \end{array} \right\}$$

Similarly,

$$\overline{R}_2 = \left\{ \begin{array}{llll} (1,0,0,0,0|0), & (1,0,0,0,0|1), & (0,0,0,1,0|2), & (0,0,0,0,1|3), \\ (0,1,0,0,0|0), & (0,0,1,0,0|0), & (0,1,0,0,0|1), & (0,0,1,0,0|1) \end{array} \right\}$$

Thus,

$$\begin{aligned} \overline{R} &= \overline{R}_1 \bowtie \overline{R}_2 \\ &= \left\{ \begin{array}{llll} (1,0,0,0,0|0,0), & (1,0,0,0,0|0,1), & (1,0,0,0,0|1,0), & (1,0,0,0,0|1,1), \\ (0,1,0,0,0|2,0), & (0,1,0,0,0|2,1), & (0,0,1,0,0|3,0), & (0,0,1,0,0|3,1), \\ (0,0,0,1,0|0,2), & (0,0,0,0,1|0,3), & (0,0,0,1,0|1,2), & (0,0,0,0,1|1,3) \end{array} \right\} \end{aligned}$$

Since $\overline{R} \bowtie [\![\neg\pi]\!] = \emptyset$, we conclude that $\text{MUX-SEM}_1 \| \text{MUX-SEM}_2 \models \pi$.

Observe that additional constraints on the initial and transition predicates are non-interfering. They merely update the augmented variables $X_A$ by the values of predicates. The following lemma hence follows from the definition.

**Lemma 1.** *Let $P = \langle X, L, I, T \rangle$ be a process, $\Psi$ a set of predicates, and $\pi$ a predicate. Then $P \models \pi$ if and only if $A(P, \Psi) \models \pi$.*

*Proof.* According to Definition 2, given any trace $\alpha = s^0 s^1 \ldots s^k$ in $P$, the corresponding trace in $A(P, \Psi)$ is $\beta = t^0 t^1 \ldots t^k$, where $t^i(u_\chi) = \chi(s^i)$ for any $\chi \in \Psi$ and $s^i = t^i \downarrow_{X \cup L} (0 \leq i \leq k)$. It is easy to prove $\alpha \models \pi \iff \beta \models \pi$. So we can conclude that $(P \models \pi) \Leftrightarrow (A \models \pi)$. □

The main process for thread-modular model checking with iterative refinement is shown in Algorithm 2. Given $N$ processes $P_1, P_2, \cdots, P_N$ and a predicate $\pi$, the algorithm decides is $\pi$ satisfied on the whole system or not. In lines 3-8, the algorithm performs the regular thread-modular model checking. Then it analyzes is there any initial state in the reachable set of error states. If so, it reports "FAILURE". Otherwise, it calls a subroutine to refine the model.

**Input**: $P_j = \langle X, L_j, I_j, T_j \rangle$ : a process for $1 \leq j \leq N$; $\pi$ : a predicate over $St[X \cup L_1 \cup \cdots \cup L_N]$
**Output**: "*PASS*" or "*FAILURE*"
1 $\mathit{error} \leftarrow [\![\neg\pi]\!]$;
2 $\Psi_j \leftarrow \emptyset$, for $j = 1, \ldots, N$; `// the essential predicate set for` $P_j$
3 **repeat**
4     **foreach** $j = 1, \ldots, N$ **do**
5         $\overline{R}_j \leftarrow \mathit{Re}[G(P_1) \| \cdots G(P_{j-1}) \| P_j \| G(P_{j+1}) \| \cdots \| G(P_N)]$;
6     **end**
7     $\overline{R} \leftarrow \overline{R}_1 \bowtie \overline{R}_2 \bowtie \cdots \bowtie \overline{R}_N$;
8     **if** $\overline{R} \bowtie \mathit{error} = \emptyset$ **then**
9         **return** *PASS*;
10     **if** $\overline{R} \bowtie \mathit{error} \bowtie [\![I_1]\!] \bowtie \cdots \bowtie [\![I_N]\!] \neq \emptyset$ **then**
11         **return** *FAILURE*;
    `// refine` $P_1, P_2, \ldots, P_N$ `by` $\overline{R}$ `and` *error*
12     $\mathit{refinable} \leftarrow \mathit{Refine}(\overline{R}, \mathit{error}, P_1, P_2, \ldots, P_N, \Psi_1, \ldots, \Psi_N)$;
13     **if** $\neg\mathit{refinable}$ **then**
14         **return** *PASS*;
15 **until** *forever* ;

**Algorithm 2**: Thread-Modular Model Checking with Refinement

Algorithm 3 gives the subroutine for refining a model. For each state $s \in \overline{R} \bowtie \mathit{error}$, the algorithm tries to find the distinct essential predicate for each process. If successes, it refines the component model using these found predicates. Otherwise, it adds the predecessors of $s$ into *error*.

**Input**: $\overline{R}$ : a state set; *error* : a state set; $P_j = \langle X, L_j, I_j, T_j \rangle$ : a process for $1 \le j \le N$; $\Psi_1, \cdots, \Psi_N$
**Output**: $\top$ if any of the processes is refined; $\perp$ otherwise

*refined* $\leftarrow \perp$;
$S \leftarrow \overline{R} \bowtie$ *error*;
**while** $S \neq \emptyset$ **do**
    *predicateAdded* $\leftarrow \perp$;
    remove an $s$ from $S$;
    **foreach** $j = 1, \ldots, N$ **do**
        $\Psi_j^s \leftarrow \{\chi_l^s : \chi_l^s$ is a distinct essential predicate from all $\chi \in \Psi_j\}$;
        **if** $\Psi_j^s \neq \emptyset$ **then**
            $P_j, \Psi_j \leftarrow A(P_j, X_{\Psi_j^s}, X_{\Psi_j^s}, \Psi_j^s), \Psi_j \cup \Psi_j^s$;
            **foreach** $i \neq j$ **do**
                $P_i \leftarrow A(P_i, X_{\Psi_j^s}, \emptyset, \emptyset)$;
            **end**
            *refined*, *predicateAdded* $\leftarrow \top, \top$;
            break;
    **end**
    **if** $\neg$*predicateAdded* **then**
        // $\langle X, L, I, T \rangle = P_1 \| P_2 \| \cdots \| P_N$ , $W = X \cup L$
        *pre* $\leftarrow \{ {}'s \downarrow_W : T({}'s, s) = \top \}$;
        **if** *pre* $\setminus$ *error* $\neq \emptyset$ **then**
            *refined*, *error* $\leftarrow \top$, *error* $\cup$ *pre*;
**end**
**return** *refined*;

**Algorithm 3**: *Refine*$(\overline{R}, error, P_1, \ldots, P_N)$

**Lemma 2.** *Let $P_j = \langle X, L_j, I_j, T_j \rangle$ for $j = 1, \ldots, N$, and $\pi$ a predicate. For any system state $s$ in $P_1 \| P_2 \| \cdots \| P_N = \langle X, L, I, T \rangle$, when Algorithm 2 terminates, we have*

1. *$s \not\models \pi$ implies $s \in error$;*
2. *$s \in error$ implies there is a sequence $s^i = s, s^{i+1}, \ldots, s^n$ such that $s^n \not\models \pi$ and $T(s^k, s^{k+1}) = \top$ for every $i \le k < n$.*

*Proof.* (1) Note all states in $[\![\neg\pi]\!]$ are added to *error* in the begining of the algorithm; (2) Note *error* contains only states in $[\![\neg\pi]\!]$ and their predecessors. $\square$

**Lemma 3.** *Let $P_j = \langle X, L_j, I_j, T_j \rangle$ for $j = 1, \ldots, N$, and $\pi$ a predicate. Then $Re[P_1 \| P_2 \| \cdots \| P_N] \subseteq \overline{R}$ at line 7, Algorithm 2.*

*Proof.* According to Definition 1, $G(P_j)$ simulates $P_j$ for $j = 1, \ldots, N$. Then we can conclude $G(P_1) \| \cdots \| G(P_{j-1}) \| P_j \| G(P_{j+1}) \| \cdots \| G(P_N)$ simulates $P_1 \| \cdots \| P_N$ for $j = 1, \ldots, N$. So $\overline{R}_j$ is an over-approximation of $Re[P_1 \| \cdots \| P_N] \downarrow_{X \cup L_j}$ for $j = 1, \ldots, N$. Then the conclusion holds. $\square$

**Theorem 1.** *Let $P_j = \langle X, L_j, I_j, T_j \rangle$ for $j = 1, \ldots, N$, and $\pi$ a predicate.*

1. *If Algorithm 2 returns "PASS", then $P_1 \| P_2 \| \cdots \| P_N \models \pi$;*
2. *If Algorithm 2 returns "FAILURE", then $P_1 \| P_2 \| \cdots \| P_N \not\models \pi$.*

*Proof.* (1) If Algorithm 2 returns "PASS" from line 11, with the precondition $\overline{R} \bowtie \mathit{error} = \emptyset$ and Lemma 3, we get the conclusion immediately. Otherwise, if Algorithm 2 returns "PASS" from line 16, the model cannot be refined anymore. Proof by contradiction, suppose $P_1 \| P_2 \| \cdots \| P_N \not\models \pi$, then there must be a state $s \in \mathit{error}$ and it is reachable from an initial system state $s^0$. Since $s^0 \notin \mathit{error}$ (otherwise the Algorithm 2 returns "FAILURE" from line 13), there must be two adjacent states $s^i, s^{i+1}$ along the trace from $s^0$ to $s$, such that $s^i \notin \mathit{error}$ and $s^{i+1} \in \mathit{error}$. According to Algorithm 3, the state $s^i$ should be added into *error*, which means the model is refinable. This is contradictory with the assumption. (2) According to Lemma 2, if $\overline{R} \bowtie \mathit{error} \bowtie [\![I_1]\!] \bowtie \cdots \bowtie [\![I_N]\!] \neq \emptyset$, then $\exists s^0 \in [\![I_1]\!] \bowtie \cdots \bowtie [\![I_N]\!], \exists \alpha = s^0 \ldots s^k \in \mathit{Tr}[P_1 \| \cdots \| P_N] : s^k \not\models \pi$, which means $(P_1 \| \cdots \| P_N) \not\models \pi$. □

**Theorem 2.** *Let $P_j = \langle X, L_j, I_j, T_j \rangle$ for $j = 1, \ldots, N$, and $\pi$ a predicate. Algorithm 2 always terminates.*

*Proof.* In each refinement iteration, either some new states are added to the *error* set, or the system is augmented by some new predicates. Note the state space of the system is finite, the number of possible predicates is also finite (each predicate corresponds to a subset of the states). In the worst case that the algorithm cannot give conclusive answer in all iterations, it finally terminates for no new state or new predicate can be found. □

**Theorem 3.** *Let $P_j = \langle X, L_j, I_j, T_j \rangle$ for $j = 1, \ldots, N$, and $\pi$ a predicate.*

1. *If $P_1 \| P_2 \| \cdots \| P_N \models \pi$, then Algorithm 2 returns "PASS";*
2. *If $P_1 \| P_2 \| \cdots \| P_N \not\models \pi$, then Algorithm 2 returns "FAILURE".*

*Proof.* (1) According to the second statement of Theorem 1, if $P_1 \| \cdots \| P_N \models \pi$, Algorithm 2 cannot return with "FAILURE". According to Theorem 2, Algorithm 2 always terminates. Thus, if $P_1 \| \cdots \| P_N \models \pi$, the algorithm can only terminate with "PASS". (2) Similarly, according to the first statement of Theorem 1, and Theorem 2, if $P_1 \| \cdots \| P_N \not\models \pi$, the algorithm can only terminate with "FAILURE". □

## 5 Experiments

We implemented our thread-modular model checking algorithm with iterative refinement (TMMCIR) in NuSMV. For comparison, several model checking algorithms are implemented as well. They are asynchronous forward reachability (AFR) and thread-modular model checking (TMMC). To compare with SPLIT[3], we configure the tool to use the CUDD package. All benchmarks are downloaded from [18] and conducted on an 2.0GHz Intel T6400 CPU with 2GB memory.

Table 1 shows the experimental results for the simple mutual exclusion protocol MUX-SEM in Fig. 1. In the table, the column “method” shows the name of the model checking algorithm (TMMCIR, SPLIT, AFR, NuSMV, or TMMC). The number of processes instantiated in MUX-SEM is shown in the column “processes.” The time needed for verification is indicated by the column “time.” The column “BDD’s” shows the peak number of BDD nodes required. The number of refinement applied in TTMCIR and SPLIT is shown in the column “refinement.” The column “preds” gives the number of essential predicates added during verification. Finally, the column “conclusive?” shows whether the verification result is conclusive.

| method | processes | time | BDD’s | refinement | preds | conclusive? |
|---|---|---|---|---|---|---|
| TMMCIR | 20 | 0.064 | 45990 | 1 | 40 | Y |
| SPLIT | 20 | 0.887 | 331128 | 1 | 38 | Y |
| AFR | 20 | 0.064 | 45990 | na | na | Y |
| NuSMV | 20 | 0.144 | 141036 | na | na | Y |
| TMMC | 20 | 0.032 | 20440 | na | na | N |
| TMMCIR | 50 | 0.580 | 401646 | 1 | 100 | Y |
| SPLIT | 50 | 12.187 | 4555054 | 1 | 98 | Y |
| AFR | 50 | 3.320 | 1242752 | na | na | Y |
| NuSMV | 50 | 3.412 | 2444624 | na | na | Y |
| TMMC | 50 | 0.228 | 203378 | na | na | N |
| TMMCIR | 100 | 5.536 | 1510344 | 1 | 200 | Y |
| SPLIT | 100 | 207.233 | 57265726 | 1 | 198 | Y |
| AFR | 100 | 208.561 | 3059868 | na | na | Y |
| NuSMV | 100 | 614.806 | 4762520 | na | na | Y |
| TMMC | 100 | 4.200 | 2057907 | na | na | N |
| TMMCIR | 200 | 27.966 | 2439514 | 1 | 400 | Y |
| TMMCIR | 300 | 145.093 | 5767146 | 1 | 600 | Y |

**Table 1.** Experimental Results of MUX-SEM

For MUX-SEM (Fig. 1), thread-modular model checking does not give a conclusive verification result due to abstraction. Our algorithm (TTMCIR) clearly outperforms other complete algorithms in large cases. For 100 processes, TTMCIR takes only 5.536 seconds to conclude the verification; other algorithms require more than 200 seconds to give conclusive results. Moreover, our algorithm is able to finish cases with 200 and 300 processes in less than 2.5 minutes. Other complete algorithms fail to finish the verification within an hour.

We now consider a variant of the simple mutual exclusion algorithm called MUX-SEM-LAST (Fig. 4(a)). In the new algorithm, a new global variable *last* is added to record the last process which enters its critical section. Table 2 gives the experimental results.

global $x$: boolean initially $x = 1$
global $last$ : $\mathbb{N}$ initially $last = 0$
loop forever

$$\left[\begin{array}{l} l = 0 : \text{Non-Critical} \\ l = 1 : \text{request } x \wedge last := j \\ l = 2 : \text{Critical} \\ l = 3 : \text{release } x \end{array}\right.$$

(a) MUX-SEM-LAST$_j$

global $x$: boolean initially $x = 1$
local $counter$ : initially $counter = 0$
loop forever

$$\left[\begin{array}{l} l = 0 : \text{Non-Critical} \\ l = 1 : \text{request } x \wedge \\ \qquad counter := (counter + 1)\%M \\ l = 2 : \text{Critical} \\ l = 3 : \text{release } x \end{array}\right.$$

(b) MUX-SEM-COUNT$_j$

**Fig. 4.** MUX-SEM-LAST$_j$ and MUX-SEM-COUNT$_j$

| method | processes | time | BDD's | refinement | preds | conclusive? |
|---|---|---|---|---|---|---|
| TMMCIR | 50 | 1.548 | 1263192 | 1 | 100 | Y |
| SPLIT | 50 | 4.047 | 3219300 | 0 | 0 | Y |
| AFR | 50 | 87.297 | 2480394 | na | na | Y |
| NuSMV | 50 | 189.900 | 3113012 | na | na | Y |
| TMMC | 50 | 0.624 | 488516 | na | na | N |
| TMMCIR | 100 | 12.945 | 4143188 | 1 | 200 | Y |
| SPLIT | 100 | 36.557 | 28470876 | 0 | 0 | Y |
| AFR | 100 | >1h | - | na | na | N |
| NuSMV | 100 | >1h | - | na | na | N |
| TMMC | 100 | 6.636 | 2454844 | na | na | N |

**Table 2.** Experimental Results for MUX-SEM-LAST

Thread-modular model checking again fails to verify the property conclusively. Our algorithm still performs better than other complete algorithms in this example. The SPLIT tool also performs reasonably well; it finishes the case with 100 processes in 36.557 seconds whereas forward reachability and NuSMV cannot conclude in an hour. Interestingly, the SPLIT tool is able to prove the result without any refinement. Although TMMCIR requires one refinement and adds 100 essential predicates, the algorithm still concludes the verification with less time and space than SPLIT. This suggests the overhead of the proposed refinement technique is insignificant in this example.

We now consider another variant of the simple mutual exclusion algorithm (Fig. 4(b)). In MUX-SEM-COUNT, a local counter is added to each process. When a process enters its critical section, the local counter is incremented by one (modulo a constant $M$). Thread-modular model checking fails to give any conclusive result in this example. Thanks to abstraction, our algorithm and SPLIT can verify all cases in seconds. In comparison, forward reachability and NuSMV need more than 20 minutes to finish the case with 20 processes (Table 3).

For the bakery algorithm, thread modular model checking is able to verify the property conclusively (Table 4). It therefore attain the best performance with the larger case with 8 processes. Our algorithm is slightly slower (.466 seconds) than the incomplete algorithm and finishes the verification of the same case in

| method | processes | time | BDD's | refinement | preds | conclusive? |
|---|---|---|---|---|---|---|
| TMMCIR | 10 | 0.020 | 14308 | 1 | 20 | Y |
| SPLIT | 10 | 0.321 | 100019 | 1 | 18 | Y |
| AFR | 10 | 5.996 | 617288 | na | na | Y |
| NuSMV | 10 | 2.288 | 1030176 | na | na | Y |
| TMMC | 10 | 0.016 | 10220 | na | na | N |
| TMMCIR | 20 | 0.104 | 122640 | 1 | 40 | Y |
| SPLIT | 20 | 2.520 | 930020 | 1 | 38 | Y |
| AFR | 20 | 1584.179 | 2634716 | na | na | Y |
| NuSMV | 20 | 3914.385 | 159986 | na | na | Y |
| TMMC | 20 | 0.044 | 47012 | na | na | N |

**Table 3.** Experimental Results for MUX-SEM-COUNT

less than a half minute. The SPLIT tool is able to prove the same property in less than 1.5 minutes. Conventional forward reachability and NuSMV require more than 6 and 21 minutes to obtain the verification result respectively.

| method | processes | time | BDD's | refinement | preds | conclusive? |
|---|---|---|---|---|---|---|
| TMMCIR | 4 | 0.100 | 96068 | 0 | 0 | Y |
| SPLIT | 4 | 0.267 | 218708 | 0 | 0 | Y |
| AFR | 4 | 0.084 | 91980 | na | na | Y |
| NuSMV | 4 | 0.140 | 106288 | na | na | Y |
| TMMC | 4 | 0.100 | 96068 | na | na | Y |
| TMMCIR | 8 | 26.246 | 2389436 | 0 | 0 | Y |
| SPLIT | 8 | 75.141 | 26776400 | 0 | 0 | Y |
| AFR | 8 | 240.555 | 4258674 | na | na | Y |
| NuSMV | 8 | 1282.984 | 25237268 | na | na | Y |
| TMMC | 8 | 25.780 | 2389436 | na | na | Y |

**Table 4.** Experimental Results for the Bakery Algorithm

Finally, we consider the dining philosopher problem (Table 5). Thread-modular model checking cannot give conclusive answers. Most interestingly, conventional forward reachability algorithm is most efficient in this example. It takes less than 3 seconds to prove the property in the case with 10 processes. NuSMV is about 1 second slower than forward reachability. In comparison, our algorithm and the SPLIT tool require several refinements to conclude the verification. In the case with 8 processes, TMMCIR adds 20 essential predicates in 3 refinements; SPLIT adds 11 essential predicates in 6 refinement. Subsequently, both are significantly inefficient than conventional algorithms. Our algorithm requires about 16 seconds to finish whereas SPLIT takes more than 80 minutes.

In our experiments, TMMC does not give conclusive results in all examples but the bakery algorithm. If an example needs no refinement, our algorithm and

| method | processes | time | BDD's | refinement | preds | conclusive? |
|---|---|---|---|---|---|---|
| TMMCIR | 6 | 0.104 | 85848 | 3 | 12 | Y |
| SPLIT | 6 | 0.320 | 158410 | 3 | 6 | Y |
| AFR | 6 | 0.016 | 14308 | na | na | Y |
| NuSMV | 6 | 0.036 | 17374 | na | na | Y |
| TMMC | 6 | 0.008 | 7154 | na | na | N |
| TMMCIR | 8 | 1.028 | 367920 | 3 | 16 | Y |
| SPLIT | 8 | 16.442 | 1176322 | 5 | 10 | Y |
| AFR | 8 | 0.236 | 243236 | na | na | Y |
| NuSMV | 8 | 0.236 | 223818 | na | na | Y |
| TMMC | 8 | 0.020 | 24528 | na | na | N |
| TMMCIR | 10 | 15.981 | 1475768 | 3 | 20 | Y |
| SPLIT | 10 | 5274.488 | 4193266 | 6 | 11 | Y |
| AFR | 10 | 2.592 | 1815434 | na | na | Y |
| NuSMV | 10 | 3.556 | 1739444 | na | na | Y |
| TMMC | 10 | 0.052 | 48034 | na | na | N |

**Table 5.** Experimental Results for Dining Philosophers

thread-modular model checking have comparable performance. In most examples, TMMCIR and SPLIT are faster than conventional forward reachability and and NuSMV. Between our algorithm and SPLIT, ours usually performs better. This is due to the fact that our algorithm computes the reachable states separately with only one process and its environment.

## 6 Conclusions

This paper uses iterative refinement to make thread-modular model checking complete. Thread-modular model checking computes the reachable states of each process with its environment—the composition of other processes' global information. With limited global information, thread-modular model checking can compute the system reachable states quickly. However, it is incomplete for many protocols, which is what we resolved by the refinement in our approach. In most examples, our approach performs substantially better than other complete verification algorithms. The main reason is that we compute the reachable states separately with only one process and its environment. In MUX-SEM with 200 processes, we only use about 27 seconds, while other approaches use more than 1 hour.

According to our experimental data, the approach about thread-modular model checking cannot give good performance when the global variables are much more than local variables. We will take abstraction for global variables to improve its efficiency in our future work.

## Acknowledgment

Thanks to Kedar Namjoshi for introducing more detail about the auxiliary variable refinement.

## References

1. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions? In: Computer Aided Verification, Springer (2001) 221–234
2. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. Logics of Programs (1982) 52–71
3. Cohen, A., Namjoshi, K., Saar, Y.: Split: A compositional ltl verifier. In: Computer Aided Verification, Springer (2010) 558–561
4. Cohen, A., Namjoshi, K.: Local proofs for global safety properties. Formal Methods in System Design **34**(2) (2009) 104–125
5. Flanagan, C., Qadeer, S.: Thread-modular model checking. Model Checking Software (2003) 624–624
6. Gu, M., Liu, Q.: Automatic compositional reasoning for multi-thread programs. In: 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD). (2011) 175 –182
7. Henzinger, T., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Financial Cryptography, Springer (2003) 262–274
8. Henzinger, T., Jhala, R., Majumdar, R.: Race checking by context inference. ACM SIGPLAN Notices **39**(6) (2004) 1–13
9. Jones, C.: Development methods for computer programs including a notion of interference. PhD thesis, PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25 (1981)
10. Jones, C.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **5**(4) (1983) 596–619
11. Lamport, L.: Proving the correctness of multiprocess programs. Software Engineering, IEEE Transactions on (2) (1977) 125–143
12. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular counterexample-guided abstraction refinement. Static Analysis (2011) 356–372
13. Misra, J., Chandy, K.: Proofs of networks of processes. Software Engineering, IEEE Transactions on (4) (1981) 417–426
14. Namjoshi, K.: Symmetry and completeness in the analysis of parameterized systems. In: Verification, Model Checking, and Abstract Interpretation, Springer (2007) 299–313
15. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Communications of the ACM **19**(5) (1976) 279–285
16. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. Tools and Algorithms for the Construction and Analysis of Systems (2001) 82–97
17. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: International Symposium on Programming, Springer (1982) 337–351
18. SPLIT: (http://split.ysaar.net/)